A Mechanical Proof of Segall's PIF Algorithm

Hesselink, Wim H.

Published in:
Formal Aspects of Computing

Publication date:
1997

Formal Aspects of Computing (1997) 9: 208–226

Formal Aspects of Computing

A Mechanical Proof of Segall's PIF Algorithm

Wim H. Hesselink

Dept. of Mathematics and Computing Science, Rijksuniversiteit Groningen, The Netherlands

Keywords: Distributed algorithm; Asynchrony; Message; Theorem prover; Invariant; Graph

Abstract. We describe the construction of a distributed algorithm with asynchronous communication together with a mechanically verified proof of correctness. For this purpose we treat Segall's PIF algorithm (propagation of information with feedback). The proofs are based on invariants, and variant functions for termination. The theorem prover NQTHM is used to deal with the many case distinctions due to asynchronous distributed computation. Emphasis is on the modelling assumptions, the treatment of nondeterminacy, the forms of termination detection, and the proof obligations for a complete mechanical proof. Finally, a comparison is made with (the proof of) the minimum spanning tree algorithm of Gallager, Humblet, and Spira, for which the technique was developed.

1. Introduction

The purpose of this paper is to present a mechanically supported, verified design of Segall's PIF algorithm and its extension to a distributed summation algorithm, cf. [Vaa95]. PIF stands for Propagation of Information with Feedback. The summation algorithm determines the sum of values that reside in the nodes in a network of processes.

It is well-known from sequential programming that the design of a verified algorithm is easier than the post–hoc verification of a given algorithm. It is our experience that, similarly, the right way to construct a mechanical proof of a (distributed) algorithm is to use the theorem prover from the start, i.e., to develop a verified design from scratch, possibly inspired by an existing algorithm.

An important aspect is that ghost variables (also called auxiliary variables

Correspondence and offprint requests to: Wim H. Hesselink, http://www.cs.rug.nl/~wim; email: wim@cs.rug.nl, Dept. of Mathematics and Computing Science, Rijksuniversiteit Groningen, Postbox 800, 9700 AV Groningen, The Netherlands.

or history variables) are not added later on, for the purpose of the proof, but that they appear early in the design as ordinary variables, and that they are later removed from the algorithm proper. We prefer the term "ghost variables" since that suggests their absence from the algorithm (many variables are auxiliary in some other sense; we would like to speak of history variables only if they are used to prove requirements on histories or execution sequences).

This note further serves as a simple and small example of the technique we developed and used in the mechanical proof [Hes96] of the algorithm of Gallager, Humblet, and Spira [GHS83] for the distributed determination of the minimum–weight spanning tree of a graph of processes.

In Section 2, we treat the model of asynchrony, which is completely the same as in the case of [Hes96]. In Section 3, we give the PIF algorithm and prove that it terminates in the sense that the number of messages that can be accepted during the algorithm is bounded. In Section 4, we show that, in the final state, the pointers *parent* have been set in such a way that node *root* is the "ancestor" of all other nodes.

Central concepts of the PIF algorithm are local and global termination detection. Local termination detection means that a process "knows" that it may stop executing without danger to the algorithm. Global termination detection means that some specified process (here *root*) "knows" that every process may stop executing. These concepts and the corresponding proofs are treated in Section 5. In this section we also reduce the private variables *explist.q* to ghost variables. In Section 6, we transform the algorithm into an algorithm to determine the sum of values that reside in the nodes of the graph.

We now turn to the aspects of using a mechanical theorem prover. Since it is intended as a certificate, a mechanical proof must allow an easy separation between what it proves and how it proves this. The main theorems proved must therefore be understandable from a small set of preliminaries. Moreover, the list of proof obligations must be discussed separately to ascertain that it comprises all that must be proved.

Most proof obligations have the form $A \Rightarrow B$. Such an assertion is useless, however, if it is not clear that A is satisfiable. Therefore, the author of a mechanical proof must make it as easy as possible to see that the results are nonvoid.

In particular, if we discuss a nondeterministic algorithm and we want to prove that the final state satisfies some postcondition, we have to guarantee the existence of a final state. This may not be obvious, if the algorithm is given as a binary relation between initial and final states. We therefore prefer to model all nondeterminacy by means of an oracle, which is a free variable ranging over a nonempty set. So the final state is a function of the initial state and the oracle. This has the drawback that it may not be clear that the oracle can exploit all nondeterminacy allowed by the nondeterminate description of the algorithm, but at least the danger that the author cheats himself is reduced.

We model the execution of an asynchronous distributed algorithm by a sequence of actions that consist of accepting some enabled message. For the distributed algorithm PIF we therefore define an NQTHM function `pif` such that `(pif n ora g x)` is the new global state after `n` atomic steps if `x` is the start state, `g` is the graph of processes, and `ora` is an oracle that guides the `n` subsequent choices of enabled messages. One of the proof obligations is the assertion that some global invariant `inv` is preserved. This is a theorem of the form

```
(inv .. x)  ⇒  (inv .. (pif n ora g x))
```

where .. may refer to other free variables. It is important that `ora` can indeed schedule all possible sequences of steps. Therefore `ora` must be a free variable that does not occur elsewhere in the theorem. Since `ora` occurs in all theorems on `pif`, the mechanical proof only concerns the PIF algorithm scheduled in this way. The verification that every schedule is possible is left informal, but we do prove that all messages are disabled if no enabled message is found.

In Section 7, we describe the representation of the algorithm in the prover NQTHM of [BoM88]. In particular, we discuss the representation of the nondeterminacy due to the distributed and asynchronous computation. We use here the same methods as applied for the GHS–algorithm in [Hes96]. We then proceed to construct the initial global state of the PIF–algorithm. In Section 8, we sketch the proof of invariance of our first invariant, (J0), and show how this contributes to the global invariant. We also describe our mechanical treatment of connectedness in graphs.

Finally, in Section 9, we give the list of proof obligations for the mechanical proof. This is the most interesting part of the investigation. The specification of distributed algorithms is often delicate and even the simple PIF–algorithm had some surprises for us. We come to eight proof obligations: invariance, termination, initialization, three theorems for local termination detection, one for global termination detection, and finally the correctness of the summation.

The input to the prover for the algorithms PIF and GHS consists of the event files in the directories `pif` and `ghs` of our WWW–site.

2. Modelling Asynchrony

We need to go into the modelling assumptions. Every process has a private state consisting of a number of private variables. Processes can send messages to neighbour processes. A process acts only when it accepts a message. Every message has a key word and a number of arguments. Via the declaration of the algorithm, the key word and the arguments determine the enabling condition of the message and the associated command. The enabling condition is the precondition for acceptance. The command can only contain instructions to inspect and modify private variables and to send messages to neighbour processes; it always terminates.

All processes concurrently execute the sequential program

while *true* **do**
  wait to accept some enabled message m ;
  execute the command of m
od

Since the effect of a message only depends on the message and private state of the accepting process at the moment of acceptance, we may regard the body of the above loop as one atomic step of the algorithm.

The only fairness assumption is that, whenever the set of enabled messages is nonempty, one will be accepted eventually. More formally, in the model of the algorithm, every step consists of the choice of an enabled message together with the acceptance of this message by its destination process. The algorithm terminates when all messages in transit are disabled. In this model, the global state of the system consists of the private states of the processes together with the bag of messages that are in transit (sent, but not yet accepted by the destination

process). This model of concurrency turns out to be simpler than the models for synchronous communication, and perhaps even simpler than the model with shared variables (compare [ApO91]). It is related to the I/O automata of (e.g.) [Lyn89] and to the receptive processes of [Jos92]. The model is more complex than UNITY, cf. [ChM88]. It may be regarded as a special case of UNITY, but the command associated to a message is typically much more complex than is usual in UNITY programs.

For the formal description of the global state, we introduce variables *buf.q* to hold the bag of messages in transit to process q. So, if process p sends a message with key word *kw* and arguments a to process $q \neq p$, according to the command *send*(q, kw, a), this has the effect

$$buf.q := buf.q + \{(kw, a)\}$$

where $+$ denotes bag addition. Process q can accept any enabled message $m \in buf.q$. Acceptance of m has the effect

$$buf.q := buf.q - \{m\}$$

followed by execution of the command associated to m.

We use two other sending commands. Firstly, a multicast to a set S of destinations is expressed by *mcast*(S, kw, a), which is equivalent to

for all $r \in S$ **do** *send*(r, kw, a) **od**

Secondly, in order to allow a finer grain of atomicity and some separation of concerns, we introduce the possibility that a process sends a message m to itself by means of the command *delay*(m). The purpose of selfmessages is to postpone the execution of an action until execution is appropriate. Since the reasons to *send* and to *delay* are quite different, we take command *send*(p, kw, a) for process p itself to be equivalent to *skip*. This makes some invariance proofs easier.

In order to discuss the messages in transit, we introduce the notations

$$kw \textbf{ at } q \quad \equiv \quad (\exists\, a :: (kw, a) \in buf.q)$$
$$(kw, r) \textbf{ at } q \quad \equiv \quad (\exists\, b :: (kw, r, b) \in buf.q)$$

which express that some message is in transit to q with key word *kw* (and first argument r, etc.). We write **not-at** for the negation of **at**. So u **not-at** q stands for $\neg(u \textbf{ at } q)$. If we want to discuss the number of such messages instead of the existence, the operator **at** is replaced by #. So, for example, $(kw, r)\#q$ stands for the number of messages in transit to q with key word *kw* and first argument r.

3. The PIF Algorithm

Given is an undirected graph (V, E) without self-loops. The nodes of the graph are processes that can asynchronously send messages to neighbour processes. Processes q and r are neighbours iff $(q, r) \in E$. We write *Nhb.q* for the set of neighbours of q. Since the graph has no self-loops and is undirected we have $q \notin Nhb.q$ and

$$r \in Nhb.q \quad \equiv \quad q \in Nhb.r$$

We assume that the graph is connected and that the algorithm starts in a situation where process *root* $\in V$ has sent messages (*signal*, *root*) to all its

neighbours. Initially, the only messages in transit are these messages from *root* to its neighbours. This is captured in the initial predicate

(Init0) $buf.q = \textbf{if } q \in Nhb.root \textbf{ then } \{(signal, root)\} \textbf{ else } \emptyset \textbf{ fi}$

The purpose of the algorithm is that eventually all nodes receive signals, but that no unnecessary signals are sent. In particular, the algorithm must terminate. A secondary purpose is that process *root* eventually "knows" that all nodes have been reached. As an application we extend the algorithm to an algorithm to determine the sum of values that reside in the nodes of the graph and to collect this sum at the root. For the formal specification of this extension we refer to Section 6.

We give every node p a private variable *parent.p* to hold the name of the sender of the first *signal* that p receives. We also give every node p a private variable *explist.p* to hold the set of neighbours process p is expecting *signal*s from.

Initially, all processes $p \neq root$ have $parent.p = p$ and all processes p have $explist.p = Nhb.p$. We introduce the name *chaos* for the initial *parent* of *root* and we postulate that $chaos \notin V$. So we have the initial conditions

(Init1) $parent.q = \textbf{if } q = root \textbf{ then } chaos \textbf{ else } q \textbf{ fi}$
(Init2) $explist.q = Nhb.q$

Later *parent.p* becomes the name of the sender of the first message *signal* accepted by p. Upon acceptance of this first *signal*, process p broadcasts the *signal* to all its other neighbours. It also sends a *signal* back to the parent, but this action is delayed and made dependent on some enabling condition *Enco*. So there are two types of messages, as specified in the following declaration

```
accept (signal, j) =
    enabling true
•   explist := explist \ {j} ;
    if parent = self then
        parent := j ;
        delay(sendrep) ;
        mcast(explist, signal, self)
    fi
end

accept (sendrep) =
    enabling Enco
•   send(parent, signal, self)
end
```

A message is a list that consists of a key word followed by a number of arguments. The declaration defines, for each key word, the number of arguments, the enabling condition and the associated command. Above we declare messages with key words *signal* and *sendrep*. The enabling condition is prefixed by **enabling**. The bullet separates the enabling condition from the command. The command is expressed in an ALGOL-like language. The variables mentioned are the private variables of the accepting process, j is the input parameter, and *self* is the name of the accepting process.

Above we announced that *explist.q* should hold the set of nodes from which process q is expecting signals. This assertion is captured in the invariant, for all q and $r \in V$:

(J0) $(signal, r)$ **at** $q \Rightarrow r \in explist.q$

In the proof that, indeed, predicate (J0) is invariant, we use the invariants

(J1) $parent.q = q \wedge r \in explist.q \Rightarrow q \in explist.r$
(J2) $sendrep$ **at** $q \Rightarrow q \in explist.(parent.q)$
(J3) $(signal, r)\#q \leqslant 1$

The free variables q and r in the invariants always range over V. In particular they differ from *chaos*.

The proof of invariance of (J0) goes as follows. If process $p \neq q$ accepts a message, it only threatens (J0) by sending $(signal, p)$ to q. This threatens (J0) for $r = p$. If p accepts the message *signal*, preservation of (J0) follows from (J1). If p accepts *sendrep*, preservation of (J0) follows from (J2). Predicate (J0) is also threatened if q deletes r from *explist.q*. This only happens when q accepts the message $(signal, r)$. Then q removes this message from *buf.q*. Therefore, in this case, preservation of (J0) follows from (J3).

For the proofs of (J1), (J2), (J3), we need

(J4) $(signal, q)$ **at** $r \Rightarrow parent.q \neq q$
(J5) $q \notin explist.q$
(J6) $(signal, q)$ **at** $parentq \Rightarrow sendrep$ **not-at** q

Finally, for the proof of (J6) we need

(J7) $sendrep\#q \leqslant 1$

More precisely, preservation of (J1) follows from (J4); preservation of (J2) follows from (J0), (J1), (J5), and (J6); preservation of (J3) follows from (J4) and (J6); preservation of (J4) follows from (J0), (J2), and (J5); preservation of (J5) is trivial; preservation of (J6) follows from (J4) and (J7); preservation of (J7) follows from (J2) and (J5). Notice that cyclic dependencies are allowed here. In fact, we assume that all these predicates hold in the precondition of a step, and then we prove that they all hold in the postcondition.

The word invariant may give rise to misunderstanding. In the implicit physical model the actions of the processes may overlap and the invariants need almost never hold. Indeed, the invariants only refer to the mathematical model with the grain of atomicity as specified. The precise definition requires the following definition of "reachable state". A state of the algorithm consists of the values of the private variables of the processes together with the bag of messages in transit. Every atomic action of a process is a transition from one state to another. An execution of the algorithm is a sequence of transitions that starts in some initial state. A state is called *reachable* if it occurs in an execution. Finally, an *invariant* is defined to be a predicate that holds in all reachable states.

We also need a method to verify invariants. So we have to provide a proof theory. Following [Tel94], we write $\{P\} \rightarrow \{Q\}$ to denote that every atomic action of the algorithm that starts in a state where P holds, terminates in a state where Q holds. We define a predicate P to be a *strong invariant* if it holds initially and satisfies $\{P\} \rightarrow \{P\}$. Notice that Tel ([Tel94] p. 51) uses the term *invariant* where we use the term *strong invariant*.

It is easy to see (cf. theorem 2.11 of [Tel94]), that every predicate implied by a strong invariant is an invariant according to our definition. This is the only way we prove invariance of predicates. So, alternatively, we might define an invariant to be a predicate that is implied by a strong invariant.

For the mechanical proof of termination we define

$$vfloc.q \quad = \quad \#explist.q + \#(parent.q = q) + \#(sendrep \textbf{ at } q)$$

where $\#S$ is the number of elements of a set S, whereas for P boolean, $\#P$ is 0 or 1 if P is false or true, respectively. It follows from (J0), (J2), (J5), and (J7) that *vfloc.q* decreases with one whenever process q accepts a message. It is clear that *vfloc.q* does not change if $p \neq q$ accepts a message. It follows that $vf = \sum_q vfloc.q$ decreases with one whenever some process accepts a message. Initially, we have $vf \leqslant (\#V)^2$. Since vf remains $\geqslant 0$, it follows that after a bounded number of actions all messages are disabled. This proves that the algorithm terminates.

4. The Manner of Termination

For a more careful discussion of termination we introduce the predicate *Dis.q* to express that all messages in transit to process q are disabled. So we have

$$Dis.q \quad = \quad signal \textbf{ not-at } q \quad \wedge \quad \neg(sendrep \textbf{ at } q \wedge Enco.q)$$

Let *DIS* be the predicate that all messages in transit are disabled. So

$$DIS \quad = \quad (\forall\, q :: Dis.q)$$

Above we proved that, after a bounded number of actions, predicate *DIS* holds.

In order to prove that the algorithm does something useful, we postulate the invariants

(K0) $parent.q = q$
$\quad \vee \quad (\exists\, n :: parent^n.q = root \quad \wedge \quad (\forall\, i : 0 \leqslant i \leqslant n : parent^i.q \in V))$

(K1) $r \in Nhb.q \quad \wedge \quad parent.r = r \quad \Rightarrow \quad parent.q = q \quad \vee \quad (signal, q)$ **at** r

Preservation of (K0) follows from (J4). Preservation of (K1) follows from (J4) and the new postulate

(K2) $parent.q = q \quad \Rightarrow \quad explist.q = Nhb.q$

Preservation of (K2) is trivial.

Since *chaos* is the initial *parent* of *root* and *parent* is only modified under the precondition $parent = self$, we clearly have the invariant

(K3) $parent.root = chaos$

It now follows from (K1) that

$$Dis.r \quad \wedge \quad r \in Nhb.q \quad \wedge \quad parent.r = r \quad \Rightarrow \quad parent.q = q\ .$$

Since the graph is connected, induction over the graph with this property and (K3) implies that, for all nodes q,

(D0) $DIS \quad \Rightarrow \quad parent.q \neq q$

Together with (K0) this yields

(D1) $DIS \Rightarrow (\exists\, n :: parent^n.q = root \wedge (\forall\, i : 0 \leqslant i \leqslant n : parent^i.q \in V))$

This shows that, in the final state, the pointers *parent* form a spanning tree of the graph.

Notice that, up to now, all results are independent of the enabling condition *Enco*. So, we may choose *Enco* equal to false and disable message *sendrep* forever. This is equivalent to the removal of *sendrep*. We thus get Segall's first algorithm PI (propagation of information without feedback). Indeed, if one wants to use this algorithm to propagate information, one can just add the information to message *signal* as a second parameter.

5. Termination Detection

Let us define the term *local function* at a process q to mean a function of the private state of process q, which may also involve pending selfmessages of q (but no other messages in transit). So, anthropomorphically speaking, process q knows the values of its local functions.

In Segall's PIF algorithm, the purpose of feedback is local and global termination detection. We define *local termination detection* to mean the existence of local functions *locterm.q* at q, which eventually become true and are such that *locterm.q* implies that no more messages will arrive at q. Since, as we have proved, *DIS* holds after a finite number of actions, the proof obligations for local termination detection are

(a) $DIS \Rightarrow locterm.q$
(b) *locterm.q* is stable (once true, it remains true)
(c) $locterm.q \Rightarrow kw$ **not-at** q

Property (c) expresses that, if *locterm.q* holds, there are no messages in transit to q. This implies *Dis.q*, but it also implies that there are no disabled pending messages.

For the PIF algorithm we define *global termination detection* to mean the existence of a specified process q_0 such that *locterm.*q_0 implies termination. We shall take $q_0 = root$. So, the proof obligation will be

(d) $locterm.root \Rightarrow DIS$

For the purpose of local termination detection we define

$$locterm.q = (explist.q = \emptyset \wedge sendrep \textbf{ not-at } q)$$

The properties (b) and (c) are easy consequences of the invariant (J0). So, it remains to prove property (a). For this purpose we postulate the invariants

(K4) $q \in explist.r \wedge (signal, q)$ **not-at** $r \Rightarrow parent.q \in \{q, r\}$
(K5) $explist.q \subseteq Nhb.q$
(K6) $q \in explist.(parent.q) \Rightarrow sendrep$ **at** $q \vee (signal, q)$ **at** $parent.q$

Preservation of (K4) follows from (J5), (K2), and (K5). Preservation of (K5) is trivial. Preservation of (K6) follows from (J2) and (J5). For (K6), we postulate that, initially and always, $explist.chaos = \emptyset$. Then (K4), (K5), and (K6) hold initially.

It follows from (K4) and (D0) that

(D2) $DIS \ \wedge \ q \in explist.r \ \Rightarrow \ parent.q = r$

Together with (K6) and *Dis.q*, this implies

$$DIS \ \wedge \ q \in explist.r \ \Rightarrow \ \neg Enco.q$$

We now assume, for all processes q, that

(*) $explist.q = \emptyset \ \Rightarrow \ Enco.q$

Then we get

(D3) $DIS \ \wedge \ q \in explist.r \ \Rightarrow \ explist.q \neq \emptyset$

Now assume that *DIS* holds and $explist.q \neq \emptyset$. Let us define a *parent* path to be a sequence $(q_0, \ldots, q_n)$ of elements of V such that $q_i = parent.q_{i+1}$ for all $i < n$. It follows from (D1), (K3), and $chaos \notin V$, that every *parent* path satisfies $n + 1 \leqslant \#V$. On the other hand, repeated application of (D3) enables us to construct an infinite sequence $q = q_0, q_1, \ldots$ such that $q_{i+1} \in explist.q_i$ for all $i \geqslant 0$. By (D2), we then have $parent.q_{i+1} = q_i$, so that the infinite sequence is a *parent* path. Since this contradicts $n + 1 \leqslant \#V$, it follows that

$$DIS \ \Rightarrow \ explist.q = \emptyset$$

Using (*) and *Dis.q*, we then obtain property (a).

For the purpose of global termination detection, we assume

(**) $Enco.q \ = \ (explist.q = \emptyset)$

Then we prove the invariance of

(K7) $q = root \ \vee \ parent.q = q \ \vee \ sendrep \ \mathbf{at} \ q \ \vee \ explist.q = \emptyset$
(K8) $q \neq root \ \Rightarrow \ parent.q \in V$

For brevity, invariants in the remainder of the paper are usually stated without detailed justification.

Predicate (K7) together with (J2) implies, for all $q \in V$,

$$q \neq root \ \wedge \ locterm.(parent.q) \ \Rightarrow \ locterm.q$$

Using (K0) and (K8) we then obtain

(D4) $locterm.root \ \wedge \ parent.q \neq q \ \Rightarrow \ locterm.q$

It follows from (K1), (J0), and (J1) that

$$r \in Nhb.q \ \wedge \ parent.r = r \ \wedge \ parent.q \neq q \ \Rightarrow \ r \in explist.q$$

Together with (D4) this implies

$$locterm.root \ \wedge \ r \in Nhb.q \ \wedge \ parent.q \neq q \ \Rightarrow \ parent.r \neq r$$

Now (K3) and connectness of the graph implies

$$locterm.root \ \Rightarrow \ parent.q \neq q$$

Together with (D4) this yields that *locterm.root* implies *locterm.q* for all q. Together with condition (c) we then get (d).

At this point we eliminate the variables *explist*, or rather we add new private variables *expcnt* of type integer and reduce *explist* to ghost variables. We do this by means of the invariant

(K9) $expcnt.q = \#explist.q$

In view of (J0) and the first assignment of *signal*, we extend the body of *signal* with the assignment

$$expcnt := expcnt - 1$$

Now indeed (K9) is invariant.

In view of (K2), the multicast in *signal* is replaced by

$$mcast(Nhb.self \setminus \{j\}, signal, self)$$

In view of (K9), the enabling condition *Enco* of *sendrep* is replaced by

(***) $Enco.q \;=\; (expcnt.q = 0)$

In this way, the only remaining occurrences of *explist* in the algorithm are in the first assignment of *signal*. Since this is an assignment to *explist* itself, indeed, variable *explist* has been reduced to a ghost variable, cf. [OwG76] (3.6). So, we can delete all occurrences of *explist* from the declarations. We shall not do so, since it requires much work to convince the theorem prover that this is allowed.

6. A Distributed Summation Algorithm

We now transform the PIF algorithm into an algorithm to determine the sum of values that reside in the nodes of the graph, cf. [Vaa95].

We introduce private variables *value.q* of type natural number for all nodes q. Let *sum* be the initial sum $\sum_q value.q$. We require that $value.root = sum$ holds when the algorithm terminates. So the required postcondition is

(e) $DIS \;\Rightarrow\; value.root = sum$

In order to collect the values at the *root*, we let the messages *signal*, when sent to the *parent*, transfer the *value* of the child to the parent. We accumulate these values in the private variable *value* of the parent. So, we give the messages *signal* a second argument u, which is 0 for the outward *signal*s and which carries the *value* of the sender for the feedback *signal*s.

In this way, the declarations of the messages become

accept $(signal, j, u) =$
  enabling *true*
• $explist := explist \setminus \{j\}$;
  $expcnt := expcnt - 1$;
  $value := value + u$;
  if $parent = self$ **then**
    $parent := j$;
    delay (*sendrep*) ;
    mcast $(Nhb.self \setminus \{j\}, signal, self, 0)$
  fi
end

accept (*sendrep*) =
  enabling $expcnt = 0$
• *send* (*parent*, *signal*, *self*, *value*) ;
  $value := 0$
end

The assignment to *value* in *sendrep* is superfluous but convenient for the proof. For the purpose of the proof we define the state functions

$$weight.q \quad = \quad value.q + (\sum\ m \in buf.q : m = (signal, -, u) : u)$$

to express the total value at or in transit to process q. Here the $\sum$-expression denotes the sum of the second arguments u of all messages m in transit to q with key word *signal*. We postulate the invariant

(L0) $\qquad (\sum\ q \in V :: weight.q) \quad = \quad sum$

If the acceptance of a message by process q modifies weights, it is the acceptance of *sendrep*. In that case, the weight of q is transferred to the parent of q. Notice that $parent.q \neq q$ then follows from (J2) and (J5). Moreover $parent.q \in V$ follows from (K8) and the new postulate

(L1) $\qquad sendrep$ **not-at** $root$

Preservation of (L1) follows from (K3).

In order to establish the postcondition, we prove the invariance of

(L2) $\qquad q = root \quad \vee \quad parent.q = q \quad \vee \quad sendrep$ **at** $q \quad \vee \quad value.q = 0$

Preservation of (L2) when q accepts *signal* follows from (K7), (K9), and (J0).

In view of (D0) and (a), predicate (L2) implies

$$DIS \quad \wedge \quad q \neq root \quad \Rightarrow \quad value.q = 0$$

It follows from (a) and (c) that

$$DIS \quad \Rightarrow \quad weight.q = value.q$$

Combining this with (L0) we obtain the required postcondition

(e) $\qquad DIS \quad \Rightarrow \quad value.root = sum$

7. Using a Theorem Prover

The arguments used above are quite detailed. They were formed and tested by frequent interaction with our theorem prover. In the remainder of this paper we describe our approach to the use of that prover and the results of the interaction.

We use the theorem prover NQTHM of [BoM88]. This prover has a variation of pure LISP as its assertion language. We use association lists to bind values to variables. If `x` is an association list, the term `(assoc a x)` is the first element `z` of list `x` with `(car z) = a`. We define a function `putassoc` such that `(putassoc b w x)` yields the modification of list `x` where value `w` is bound to key `b`. The definition is

```
(defn putassoc (b w x)
    (if (nlistp x) (cons (cons b w) nil)
        (if (equal b (caar x))
            (cons (cons b w) (cdr x))
            (cons (car x) (putassoc b w (cdr x))) ) ) )
```

After this definition, we submit the lemma

```
(lemma assoc-put (rewrite)
    (equal (assoc a (putassoc b w x))
           (if (equal a b) (cons b w)
               (assoc a x) ) ) )
```

Without hints, NQTHM is able to prove this simple lemma by induction (in the size of x). The term (rewrite) means that the prover can later use the lemma to rewrite an expression that fits the lefthand side of the equality.

We model the private state of each node q as an association list. We model the global state as an association list that associates to each node its private state. It follows that the value of private variable a of process q in global state x is given by function

```
(defn val (a q x)
    (cdr (assoc a (cdr (assoc q x)))) )
```

In particular, *buf.q* is given by

```
(defn buffer (q x)
    (val 'buffer q x) )
```

Since NQTHM has no bags, we treat (buffer q x) as an ordered list of messages, but we shall use the order of this list only after a nondeterminate permutation.

The model of distributed computations with asynchronous messages introduces the nondeterminacy that, at every step, some enabled message (if existent) is chosen to be accepted by its destination. Since *buf.q* represents the bag of messages in transit to process q, this nondeterminacy is split into two parts: at every step an enabled process must be chosen, together with an enabled message in transit to it.

Let us first describe the deterministic algorithm. We let (step p decl x) be the new global state, if process p accepts the first element of the list (buffer p x) and acts according to the declaration decl of the messages.

We define a boolean function enabledany to decide whether there is an enabled message. We define a function swapbufena to permute *buf.p* in such a way that an arbitrary enabled message becomes its first element. Similarly, we define a function favproc to yield an arbitrary enabled process. The first argument of these two functions is an oracle to guide the nondeterminacy. Now an arbitrary nondeterministic step is defined by

```
(defn genstep (ora plist decl x)
    (if (enabledany plist decl x)
        (let ((p (favproc (car ora) plist decl x)))
             (step p decl
                   (swapbufena (cdr ora) p decl x) ) )
        x ) )
```

The argument plist is the list of processes. Notice that the global state remains unchanged if there is no enabled message. Also notice that ora serves as the oracle of genstep and that (car ora) and (cdr ora) can take arbitrary values. In this way, if an enabled message exists, an arbitrary enabled message of an arbitrary enabled process is chosen.

We do not describe the construction of function step. It is an interpreter

for message declarations in a LISP-like syntax. The declaration of Section 6 is represented by

```
(defn dcl-pif (g)
    `((signal (j u) (true)
        ((put value (plus u value))
         (put explist (delete j explist))
         (put expcnt (sub1 expcnt))
         (if (equal parent self)
             ((put parent j)
              (delay sendrep)
              (mcast (delete j (neighbours ',g self))
                     signal self 0) ) ) ) )

      (sendrep ()
         (zerop expcnt) ; the enabling condition Enco
         ((send parent signal self (fix value))
          (put value 0) ) ) ) )
```

The argument g of the declaration is a representation of the graph. The function neighbours represents *Nhb*, which of course depends on the graph under consideration. The inverted quote, the quote, and the comma are used to import argument g into the S-expression (the reader is advised to believe this and not to ask for details).

The key word put represents assignment for the interpreter. Function delete deletes an element from a list. Function fix coerces its argument to a natural number (this is not important, but convenient for the proof). An arbitrary step of the PIF algorithm is now defined by

```
(defn genstep-pif (ora g x)
    (genstep ora (nodes g) (dcl-pif g) x) )
```

Here (nodes g) is the list of nodes of graph g. The new state when the algorithm takes n arbitrary steps is defined by

```
(defn pif (n ora g x)
    (if (zerop n) x
        (pif (sub1 n) (cdr ora) g
             (genstep-pif (car ora) g x) ) ) )
```

Again, (car ora) and (cdr ora) can take arbitrary values. In this way, an arbitrary schedule of enabled messages can be chosen.

The initial state of the PIF–algorithm is constructed as follows. For every node q we define the initial private state as the association list

```
(defn initpriv (g ora q)
    (list (cons 'buffer (if (member 'root (neighbours g q))
                            '((signal root 0))
                            nil ))
          (cons 'parent (if (equal q 'root) 'chaos q))
          (cons 'explist (neighbours g q))
          (cons 'value (fix (cdr (assoc q ora))))
          (cons 'expcnt (card-of (neighbours g q))) ) )
```

The initial global state is the association list that associates to each node q its private state:

```
(defn initlist (g ora nod)
    (if (nlistp nod) nil
        (cons (cons (car nod)
                    (initpriv g ora (car nod)) )
              (initlist g ora (cdr nod)) ) ) )

(defn initstate (g ora )
    (initlist g ora (nodes g)) )
```

Here ora is the oracle to choose the arbitrary numbers *value.q*. It follows that the initial sum of these values is equal to (sumover (nodes g) ora), where

```
(defn sumover (nod z)
    (if (nlistp nod) 0
        (plus (cdr (assoc (car nod) z))
              (sumover (cdr nod) z) ) ) )
```

We introduce a function to express the special status of the names *root* and *chaos*:

```
(defn specialnames (g)
    (and (member 'root (nodes g))
         (not (member 'chaos (nodes g))) ) )
```

8. Ingredients of Proofs

The invariant (J0) is represented by

```
(defn jq0 (q r x)
    (or (nooc1 'signal q r x)
        (member r (explist q x)) ) )
```

where (nooc1 'signal q r x) expresses that (*signal*, *r*) is not at *q*. The assertion that (J0) is invariant is contained in

```
(lemma jq0-kept-valid (rewrite)
    (implies (and (jq0 q r x)
                  (jq1 p q x)
                  (jq2 p x)
                  (kq2 g p x)
                  (jq3m p x) )
             (jq0 q r (steppif g p x)) ) )
```

where (steppif g p x) represents one deterministic step of the PIF algorithm when p accepts the first message in its buffer. The lemma is proved by distinguishing the cases $p = q$ and $p \neq q$, and also by considering the various messages that p can accept.

The functions jq1, jq2, kq2 represent (J1), (J2), (K2). The function jq3m is the corollary of (J3) stating that, if the first message in *buf.p* is (*signal*, *r*), that message does not occur in the remainder of *buf.p*.

Similar lemmas are proved for all invariants. We then form the conjunction of all invariants, and the universal quantification over all nodes q and r, to get the global invariant globinv. We then combine the lemmas for the individual invariants to a proof of invariance of globinv. This part of the proof is a kind of bookkeeping to verify that indeed all invariants have been treated.

A rather different class of problems is encountered when we must use the connectedness of the graph. We define connectedness in a graph by means of connection via a list ed of directed edges:

```
(defn connected (x y ed)
    (if (nlistp ed) (equal x y)
        (or (connected x y (cdr ed))
            (and (listp (car ed))
                 (connected x (caar ed) (cdr ed))
                 (connected (cdar ed) y (cdr ed)) ) ) ) )
```

This reads: x and y are connected via ed if ed is empty and x = y, or ed is not empty and x and y are connected via the tail of ed, or the head of ed is a pair, say (u,v), and x is connected to u and v is connected to y, in both cases via the tail of ed. We then define connectedness of graph g by requiring that *root* is connected to all nodes of g with respect to the list of edges of g. This is done by induction over the list of nodes.

Some assertions are proved by induction over the graph, according to the following theory. Assume that prop is a property of the list ed and that crit is a condition on the nodes, and assume that (prop ed) implies, that (crit u) implies (crit v) for every pair (u,v) of list ed. This assumption is formalized in:

```
(axiom prop-crit (rewrite)
    (implies (and (prop ed)
                  (listp ed) )
             (and (implies (crit (caar ed))
                           (crit (cdar ed)) )
                  (prop (cdr ed)) ) ) )
```

Then a path from q to r suffices to see that (crit q) implies (crit r), according to

```
(lemma connected-crit (rewrite)
    (implies (and (connected q r ed)
                  (prop ed)
                  (crit q) )
             (crit r) ) )
```

This theory is instantiated twice in the proof. It is also used three times in the proof of [Hes96].

9. Proof Obligations

The main body of the proof yields a global invariant globinv, which is the conjunction of the universal quantifications of the invariants in the families (J...), (K...), and (L...) over all nodes. We regard its invariance as the first proof obligation:

```
(lemma pif-preserves-globinv (rewrite)
    (implies (globinv g sum x)
             (globinv g sum (pif n ora g x)) ) )
```

As above, g is the graph and x is the global state. Variable sum represents the initial $sum = \sum_q value.q$, see the invariant (L0).

The termination theorem sketched in Section 3 now gets the form

```
(defn enabledany-pif (g x)
    (enabledany (nodes g) (dcl-pif g) x) )

(lemma pif-terminates (rewrite)
    (implies (and (enabledany-pif g (pif n ora g x))
                  (globinv g s x) )
             (lessp n (vf (nodes g) x)) ) )
```

Here, vf is the function introduced at the end of Section 3. By contraposition, this lemma says that, if globinv holds in state x and n is sufficiently large, the resulting state after n steps has no enabled messages, i.e., has terminated.

The third proof obligation is to show that the initial state satisfies globinv for the right value of s. Here we need the assumptions about *root* and *chaos*.

```
(lemma globinv-initstate (rewrite)
    (implies (specialnames g)
             (globinv g (sumover (nodes g) ora)
                      (initstate g ora) ) ) )
```

The next proof obligation is condition (a) that termination implies *locterm.q* for all nodes q. We introduce a function to combine all things known upon termination:

```
(defn finalcondition (g s x)
    (and (specialnames g)
         (globinv g s x)
         (not (enabledany-pif g x))
         (connectedgraph g) ) )
```

Now the fourth proof obligation is

```
(lemma final-implies-locterm (rewrite) ; (a)
    (implies (and (member q (nodes g))
                  (finalcondition g s x) )
             (locterm q x) ) )
```

Stability of *locterm.q* is expressed by the fifth proof obligation

```
(lemma pif-preserves-locterm (rewrite) ; (b)
    (implies (and (locterm q x)
                  (globinv g s x) )
             (locterm q (pif n ora g x)) ) )
```

The sixth proof obligation is condition (c): *locterm.q* implies that no messages are in transit to node q.

```
(lemma locterm-implies-nlistp-buffer (rewrite) ; (c)
    (implies (and (locterm q x)
                  (member q (nodes g))
                  (globinv g s x) )
             (nlistp (buffer q x)) ) )
```

Global termination detection is our seventh proof obligation

```
(lemma locterm-root-all-disabled (rewrite) ; (d)
    (implies (and (connectedgraph g)
                  (globinv g s x)
                  (locterm 'root x) )
             (not (enabledany-pif g x)) ) )
```

Finally, the goal of the summation algorithm is our eighth proof obligation

```
(lemma final-value-of-root (rewrite) ; (e)
    (implies (finalcondition g sum x)
             (equal (value 'root x) sum) ) )
```

10. Conclusions and Comparisons

Inspired by [Vaa95], we have developed an independent treatment of the PIF–algorithm using the ideas of [Hes96]. It turns out that, in comparison with [Vaa95], we have made a number of simplifications. Firstly, we assume point to point communication, cf. [WLL88]. In this way, edge names can be avoided and the associated functions `source`, `target`, `from`, and `to` disappear. Instead, we only need the function *Nhb* that yields the set of neighbours of a given node. Notice, however, that we disallow multiple edges in this way.

A related simplification is that instead of one queue of messages for every edge, we only use a bag of messages for every node. This introduces additional nondeterminacy, but the PIF algorithm remains correct. We also used this simplification in our treatment [Hes96] of the algorithm of [GHS83]; in that case the simplification requires some small modifications of the algorithm.

A further simplification is that we use completely initialized processes, just as in [WLL88]. The processes of [Vaa95] are mainly initialized by the first signal that they receive, but they have enough initial value to decide that a signal is the first one.

We do not use a prophecy variable as *tree* in [Vaa95], but only a ghost variable *explist*, which plays the same role as *rcvd* in [Vaa95]. It is instructive to see that *rcvd* indeed records aspects of the history of the node, whereas *explist* expresses that the node "must" receive signals from all its neighbours. The term "history variable" seems to have influenced its use in [Vaa95].

In the mechanical proof we have not eliminated *explist* from the algorithm, although it does not influence the computations. Since it is only a history variable in the sense of [Vaa95], the elimination of *explist* should not present any difficulties. The use and elimination of *prophecy* variables in a mechanical proof with NQTHM may be much harder.

Another difference is that the autonomous action REPORT of [Vaa95] has been replaced by our selfmessage *sendrep*. We introduced selfmessages in [Hes96] to get a better separation of concerns. Autonomous actions like REPORT also serve that purpose, but selfmessages have the advantage that they can be treated

as ordinary messages (apart from their special role in local functions) and are yet more flexible than autonomous actions.

Our treatment is more direct than the one of [Vaa95]: we first prove that a single step preserves the complete invariant and decrements the variant function. We then use induction over a sequence of steps to prove termination and the required properties of the final state. The treatment of [Vaa95] switches between the synchronic view of invariants and the diachronic view of execution sequences. We think that this requires a form of flexibility not easily realized in a mechanical proof.

Finally, our mechanical proof is complete, whereas the last results of [Vaa95] rely on proof sketches. In a research paper this is completely acceptable, but a mechanical proof requires complete clarity about what it proves and what it does not prove.

It is more difficult to compare our treatment of PIF with the treatment by Chou in [Cho95]. Where we minimize the number of ghost (or history) variables and keep the elimination of ghost variables informal, Chou's proof is based on a simulation approach. In that way he maximizes the use of history information. This requires a heavier framework, but in principle it can make the concrete proof obligations simpler. The paper [Cho95] is not detailed enough to see whether the investment pays.

Since there are now complete mechanical proofs done by the same author on the same prover of both the PIF algorithm and the GHS algorithm, we can give a rough estimate of the relative complexity of the two algorithms. It is our impression that GHS is eight times as complex as PIF. Indeed, the input to the prover is 3580 lines for PIF and around 30000 lines for GHS. There are similar ratios between the numbers of "irreducible invariants" for the two algorithms, and between the time needed to construct the proofs. In principle, such an estimate must be taken with lots of salt. In this case, however, it can be taken seriously since the two algorithms and the methods we used for them are very similar.

Some observations concerning the invariants used. Since all actions are message driven, most invariants mention messages in transit. Many of these invariants remain valid when messages (erroneously) are removed from the network: (J0), (J2), (J3), (J4), (J6), (J7), (L1). Since arbitrary message removal cannot be correct, however, we also need invariants that express the presence of messages: (K1), (K4), (K6), (K7), (L2). Most invariants are local: they only express properties of one node, or one node in relation to one of its neighbours. The only exceptions are (K0) and (L0).

Our conclusion from this project is that complete mechanical verification of distributed algorithms is feasible. It suffices to use classical methods (invariants, variant functions), a clear and convenient computational model, a powerful theorem prover, and a lot of work. Most inventiveness is needed for the choices of the invariants. Some of them are found as weakenings of the postcondition, see (K0), or as a way to express the intention of certain variables, see (J0). Other invariants are found as weakest (or convenient) preconditions of invariants postulated before.

References

[ApO91] Apt, K. R. and Olderog, E.-R.: Verification of Sequential and Concurrent Programs. Springer. 1991.

[BoM88] Boyer, R. S. and Moore, J.: A Computational Logic Handbook. Academic Press, Boston. 1988.

[Cho95] Ching-Tsun Chou: Mechanical verification of distributed algorithms in higher-order logic. *The Computer Journal*, 38: 152–161 (1995).

[ChM88] Chandy, K. M. and Misra, J.: Parallel Program Design, A Foundation. Addison-Wesley, 1988.

[GHS83] Gallager, R. G., Humblet, P. A. and Spira, P. M.: A distributed algorithm for minimum–weight spanning trees. *ACM Trans. on Programming Languages and Systems*, 5 (1983) 66–77.

[Hes96] Hesselink, W. H.: The verified incremental design of a distributed spanning tree algorithm — extended abstract. Computing Science Reports CSR 9602, Groningen 1996.

[Jos92] Josephs, M. B.: Receptive process theory. *Acta Informatica* 29 (1992) 17–31.

[Lyn89] Lynch, N. A.: Multivalued possibilities mappings. In J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.): *Stepwise Refinement of Distributed Systems.* LNCS 430, Springer. 1990, pp. 519–543.

[OwG76] Owicki, S. and Gries, D.: An axiomatic proof technique for parallel programs. *Acta Informatica* 6 1976 319–340.

[Seg83] Segall, A.: Distributed network protocols. *IEEE Transactions on Information Theory*, IT–29 (2) 23–35, January 1983.

[Tel94] Tel, G.: Distributed Algorithms. Cambridge University Press, 1994.

[Vaa95] Vaandrager, F. W.: Verification of a distributed summation algorithm. In I. Lee, S.A. Smolka (eds). *Proceedings 6th International Conference on Concurrency Theory (Concur'95).* LNCS 962, Springer. 1995, pp. 190–203.

[WLL88] Welch, J., Lamport, L. and Lynch, N.: A lattice-structured proof technique applied to a minimum weight spanning tree algorithm. In *Proceedings 7th ACM Symposium on Principles of Distributed Computing*, 1988.

Received September 1996
Accepted February 1997 by E. C. R. Hehner